The Bernoulli Generic Matrix Library

Nikolay Mateev, Keshav Pingali, and Paul Stodghill
Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract

We have implemented the Bernoulli generic programming system for sparse matrix computations. What distinguishes it from existing generic sparse matrix libraries is that we use (i) a high-level matrix abstraction for writing generic matrix programs, (ii) a low-level matrix abstraction for describing the indexing structure and properties of sparse matrices formats, and (iii) restructuring compiler technology to transform the high-level generic programs into concrete implementations that efficiently access sparse matrices using the low-level abstraction.

This paper describes the Bernoulli Generic Matrix Library (BGML). The BGML is the C++ implementation of these high-level and low-level abstractions. Within our system, it serves as the "glue" between user's sparse matrix format implementations and the restructuring sparse compiler. In this paper, we present the interfaces of the BGML and give examples of their use. Because of its role, it is critical that the BGML not impose much of an overhead on the compiler generated code. We discuss the implementation techniques that we had to use to get the most performance from the BGML. We also discuss the difficulties that we encountered in using available C++ compilers on the BGML.

1 Introduction

Generic programming is a methodology for simplifying the development of libraries in which a set of algorithms have to be implemented for many data structures. Code explosion is avoided by mandating a common API which is (i) supported by all data structures, and (ii) used to express algorithms in a *generic* data-structure neutral fashion. For example, the C++ Standard Template Library (STL) [2] uses the API of one-dimensional sequences as the interface between data structures such as arrays and lists, and algorithms such as searching and sorting. The type systems of modern languages permit the data structure implementations and generic programs to be type-checked and compiled separately; a concrete implementation is produced by linking a generic program with a particular data structure implementation.

There is however a tension in the design of generic programming API's that becomes evident in some problem domains such as sparse matrix computations. For dense matrices, highly efficient implementations of the Basic Linear Algebra Subroutines (BLAS) [9] are usually provided by hardware vendors. For sparse matrices, the problem of developing BLAS libraries is complicated by the fact that some forty or fifty *compressed formats* are used to avoid storing zeros. Many attempts at writing sparse BLAS libraries have been confounded by the code explosion problem [8, 20]. Although it appears that generic programming is the solution to this problem, it is not clear that an appropriate API can be designed for sparse matrix libraries. As we explain in [16], a high-level API that allows the programmer to express generic matrix algorithms in a natural array notation hides details of sparse matrix formats from the compiler, so performance may suffer. On the other hand, a low-level API that exposes the details of compressed formats is not suitable for writing generic programs. This problem is likely to occur in other problem domains in which data structure properties must be exploited for high performance.

In our generic programming system [16], we solve this problem by separating the API used for writing generic programs from the API used to describe data structures. In our system, generic programs are dense matrix programs; *i.e.* the generic program writer views sparse matrices as *random-access* data structures. A low-level API describes sparse matrices as *indexed-sequential-access* data structures. We use *restructuring compiler technology* to transform abstract programs written in terms of the high-level API into efficient programs which use the low-level API. In other words, our restructuring compiler "instantiates" the generic programs into efficient sparse matrix programs.

To get efficient sparse matrix code, we addressed three problems. First, we designed an appropriate low-level API for sparse matrix formats [16]. Second, we developed the restructuring compiler technology necessary to instantiate the generic (dense matrix) programs into efficient sparse matrix programs [1, 13, 24]. Finally, we needed to implement these

ideas efficiently in an existing language. We chose to use C++ as it has language features (namely, templates and inheritance) that allow us to express our API and programs concisely. We call our implementation the Bernoulli Generic Matrix Library (BGML). The BGML is the focus of this paper. It serves two purposes:

1. The BGML provides a set of interface classes for describing sparse matrix formats to the compiler.
2. The BGML provides methods for array access notation used by the high-level API. That allows generic programs to be compiled directly by a standard C++ compiler and executed.

The rest of the paper is organized as follows. In Section 2, we discuss our motivation for developing the BGML. In Section 3 we briefly describe the abstract index structure of sparse matrices used by our compiler. In Section 4 we show the C++ implementation of that index structure. We discuss our experience with C++ template instantiation and compilation in Section 5, and conclude with related work review in Section 6.

2 Motivation for the BGML

In this section, we present a high-level picture of our generic programming system and the motivation and design requirements for the BGML.

The previous implementation The previous implementation of our generic programming system described in [24] had several defects which made it difficult to use. First, the user had to write their generic matrix programs in BML, a language of our own design that was similar in spirit to F77. Not only did this require the user to learn BML, but they also had to worry about inter-language linking issues. Another defect was that in order to implement a new sparse matrix format, the user had to write a module for our restructuring compiler that implemented the format, and to link it into the compiler. This required that the user master many interfaces and implementation details of our compiler.

In addition to not being very user-friendly, our system was very large. This was primarily because our restructuring compiler was responsible for performing method inlining and subsequent optimizations. Since the sparse matrix formats "resided" in compiler modules, these were responsibilities that the compiler had to shoulder and could not rely on a backend compiler for.

Another problem with having the spare matrix formats available only as compiler modules is that the generic programs could not be executed directly without first being run through the compiler. This was because the details of the formats were only present during compile-time. Thus, the generic programs did not have semantics per se.

```
#pragma instantiate with Bernoulli
template <class T, class BASE>
void mvm(T A, BASE x[], BASE y[])
{
    for (int i=0; i<A.rows(); i++) {
        y[i] = 0;
        for (int j=0; j<A.columns(); j++)
            y[i] += A(i,j) * x[j];
    }
}

// Will be instantiated with the Bernoulli compiler.
template void mvm(csr_matrix<double> A,
                  double x[], double y[]);
```

Figure 1: Generic MVM with Instantiation

The new implementation In redesigning our system, we set the following goals.

- Our sparse compiler should work as a single tool within a suite of tools of a larger generic programming system. In particular, our sparse compiler should work cooperatively with an underlying C++ compiler. Our sparse compiler should handle the sparse matrix computations, and leave the other generic programming problems to the C++ compiler.
- The end-user of our system should be presented with a simple mechanism with which to use our sparse compiler. This means that the user should be able to implement sparse matrix formats and generic algorithms directly in C++.
- Our sparse compiler should knit implementations for sparse matrix computations that are as efficient, and hopefully more so, than those that the programmer might have written by hand.

By allowing the user to write sparse matrix formats and generic programs in C++, we believe that our system will be much easier to use. Also, by relying on an underlying C++ compiler to perform method inlining and subsequent optimization, we can greatly simplify the back-end portion of our system. However, even if the system is easy to use, people are not likely to use it if it does not generate code that is competitive with hand-written code. Thus, performance must always be an important goal in our implementation.

We are building our system as a source-to-source transformation tool. That is, the user first runs his program though our sparse compiler which instantiates some of the the template definitions. The programmer uses pragmas, as shown in Figure 1, to indicate which template definitions are to be instantiated by the sparse compiler; the rest are left untouched. The sparse compiler generates a transformed C++ program to be run through the underlying C++ compiler which performs the remaining instantiation and usual optimizations.

BGML In order to make this design possible, we have developed the Bernoulli Generic Matrix Library (BGML). The

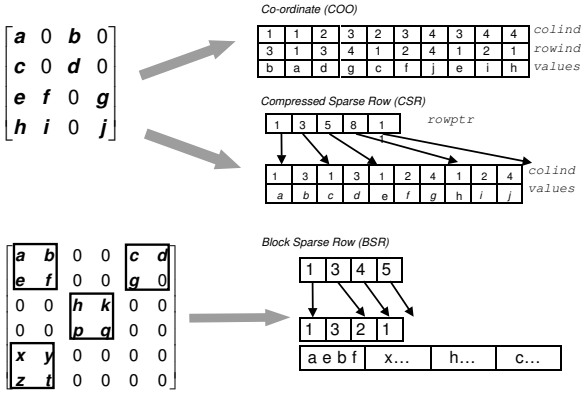

Figure 2: Compressed Formats

$$
\begin{array}{rcl}
E & : & \mathit{Index} \rightarrow E \\
 & | & \mathit{map}\{F(\mathit{in}) \mapsto \mathit{out} : E\} \\
 & | & E \oplus E \\
 & | & E \cup E \\
 & | & v \\
 & & \\
\mathit{Index} & : & \mathit{attribute} \\
 & | & < \mathit{attribute}, \ldots, \mathit{attribute} > \\
 & | & < \mathit{attribute} \times \cdots \times \mathit{attribute} >
\end{array}
$$

Figure 3: Sparse Matrix Abstraction

BGML is a library of C++ codes that serves several different functions within our system.

First, the BGML provides an "interface" between the user's sparse matrix formats and the restructuring compiler. More specifically, the user inherits from classes within the BGML (which are described in Section 4) in order to convey the relevant details of each sparse matrix format to the restructuring compiler. Put differently, the restructuring compiler can analyze the class hierarchy of each user-specified sparse matrix format in order to determine its structure and properties.

Second, the BGML gives generic programs well-defined semantics. That is, the BGML provides methods for implementing array access operations, like `A(i,j)` shown in Figure 1. By inheriting from the classes in the BGML the user's sparse matrix formats automatically provide these methods[1]. Thus, the generic programs, like the one in Figure 1, can be compiled directly by the underlying C++ compiler and executed, even without our restructuring compiler. The restructuring compiler is allowed to transform generic programs only in ways that maintain the effects of this execution.

3 Matrix Abstraction

As mentioned earlier, there are at least forty or fifty commonly used compressed formats; the NIST Sparse BLAS effort [8] supports 13 of them. Figure 2 shows a sparse matrix and three commonly used compressed formats. The simplest format is *Co-ordinate storage* (COO) in which three arrays are used to store non-zero elements and their row and column positions. The non-zeros may be ordered arbitrarily. Co-ordinate storage does not permit indexed access to either rows or columns of a matrix. *Compressed Sparse Row storage* (CSR) is a commonly used format that permits indexed access to rows but not columns. Array `values` is used to store the non-zeros of the matrix row by row, while another array `colind` of the same size is used to store the column positions of these entries. A third array `rowptr` has one entry for each row of the matrix, and it stores the position in `values` of the first non-zero element of each row of the matrix. Some of the rows of the matrix may be empty.

Some sparse matrices have small dense blocks occurring in different positions inside the matrix. It is important to exploit these dense blocks to improve storage and computational efficiency. Figure 2 shows *Block Sparse Row* (BSR) storage which can be viewed as a CSR representation in which the non-zeros are small dense blocks rather than single non-zero elements.

3.1 Index Structure

The grammar in Figure 3 is used to describe the index structure of a sparse matrix to our system [16]. The most important rule for specifying index structure is the $\mathit{Index} \rightarrow E$ (*nesting*) production rule. For example, a CSR matrix is described as $r \rightarrow c \rightarrow v$, indicating that rows must be accessed first, and within each row, elements within columns can be enumerated. The $\mathit{map}\{F(\mathit{in}) \mapsto \mathit{out} : E\}$ rule is used to describe linear and permutation transformations on the matrix indices. The $E' \oplus E''$ (*perspective*) rule means that the matrix can be accessed in different ways, using either of the index structures E' or E''. The $E' \cup E''$ (*aggregation*) rule is used to describe a matrix that is a collection of two formats, such as a format in which the diagonal elements are stored separately from the off-diagonal ones. Enumerating the elements of such matrix requires enumerating both E' and E''.

The $< \mathit{attribute}, \ldots, \mathit{attribute} >$ notation describes an index obtained from multiple co-ordinates enumerated together, as in the COO format ($< r, c > \rightarrow v$). On the other hand, $< \mathit{attribute} \times \cdots \times \mathit{attribute} >$ denotes independent indices, as in a dense matrix ($< r \times c > \rightarrow v$).

Consider the Block Sparse Row (BSR) format shown in Figure 2. Each block is accessed by a set of block indices (b_r, b_c), and the scalar elements within each block are ac-

[1] In other words, the BGML specifies the low-level API and provides a default implementation of the high-level API. The user only has to provide implementations of the low-level API in their formats.

cessed by a the offset indices (l_r, l_c). The view of BSR can be expressed as $map\{b_r * B + l_r \mapsto r, b_c * B + l_c \mapsto c : b_r \rightarrow b_c \rightarrow < l_r \times l_c > \rightarrow v\}$.

Each term E is optionally annotated with the following *enumeration properties*.

- *Enumeration order*: a description of the order in which coordinate values could be enumerated efficiently. For the CSR format above, r is random-access, and within each row, c can be enumerated efficiently in increasing order.
- *Enumeration bounds*: a description of the coordinate values that actually occur in the enumeration. A lower triangular matrix, for example, could be annotated $1 \leq c \leq r \leq$ `N`.

4 Interfaces of the BGML

Figures 4 summarizes the BGML classes that a user must inherit from in order to expose the structure and properties of their sparse matrix formats to the restructuring compiler[2]. Enumeration is supported through the use of iterators as in the STL. Enumeration order and bounds could be incorporated into the program through the use of pragmas, but we have chosen to incorporate order information into the class hierarchy by specifying different classes for enumerations that are unordered/increasing/decreasing etc. The bounds on the stored indices are conveyed to the compiler using a pragma.

4.1 Interfaces for Views

Each production in the view grammar given in Figure 3 has an associated interface, which we have implemented in the BGML as a small number of abstract base classes described in Figure 4(a). The programmer conveys views of a storage format to the sparse compiler by writing a set of classes that inherit from the appropriate interfaces.

The `term_nesting` abstract class denotes an occurrence of the $\rightarrow$ operator within the view. This abstract class takes two template parameters. The first specifies the implementation of the iterator that can be used to enumerate the index at this level. The second specifies the implementation of the substructure below this level. An implementation of CSR, in which the entries within each row are stored in order, that inherits from `term_nesting` is shown in Figure 5. `interval_iterator` and `offset_iterator` are two iterator abstract classes that are described later.

An index of the form $< r, c >\rightarrow \cdots$ is specified by inheriting from the `term_nesting` abstract class and specifying that its iterator enumerates indices of type `pair<int,int>`. This is illustrated by the implementation of Co-ordinate storage shown in Figure 6.

[2]The interfaces discussed in this paper are, in fact, simplified versions of the ones used in the actual implementation. The details that we have removed are either not important or are discussed in Section 5.

Abstract class	Methods
`term_scalar<V>`	`operator V()`
`term_nesting<I,E>`	`I begin(),I end()` `E subterm(I)`

`term_nesting2<I1,I2,E>`	`I1 begin1(),I1 end1()`
	`I2 begin2(),I2 end2()`
	`E subterm(I1, I2)`

...	

`term_map<K,E>`	`K map(E::index_type)`
	`E::index_type unmap(K)`
	`E subterm()`

`term_aggregation2<E1,E2>`	`E1 subterm1()` `E2 subterm2()`
...	
`term_perspective2<E1,E2>`	`E1 subterm1()` `E2 subterm2()`
...	

(a) Interfaces for Views

Abstract class	Methods
`unordered_iterator<K>` (no ordering)	`K operator *()` `void operator ++()`

`increasing_iterator<K>`,
`decreasing_iterator<K>`
(one-way ordering)
	`K operator *()`
	`void operator ++()`, or
	`void operator --()`

inherits from ↑ `ordered_iterator<K>` (bi-directional ordering)	

inherits from ↑
`offset_iterator<K>`
(ordered with distance)
	`int operator -(iterator)`
	`void operator +=(int)`
	`void operator -=(int)`

inherits from ↑ `interval_iterator<K>` (range of keys)	

(b) Interfaces for Iterators

Figure 4: BGML Interfaces Classes

An index like $< r \times c >\rightarrow \cdots$ has two independent iterators. To specify these sorts of views, `term_nesting2`, etc., abstract classes are provided which allow the implementation of each independent iterator to be specified. Figure 7 shows an implementation of dense matrices that uses the `term_nesting2` interface.

By a very simple analysis of these classes, the sparse compiler can infer the following relationships,

```
coo_matrix: // <r,c> -> v
    term_nesting< unordered_iterator< pair<int,int> >,
                  term_scalar<BASE> >

csr_matrix: // r -> c -> v
    term_nesting< interval_iterator<int>,
                  term_nesting< offset_iterator<int>,
                                term_scalar<BASE> > >

dense_matrix: // <r x c> -> v
    term_nesting2< interval_iterator<int>,
                   interval_iterator<int>,
                   term_scalar<BASE> >
```

which clearly indicate the nested structure of these formats, and the properties of the iterators that are used at each level.

Interfaces for expressing perspective, aggregation and map are also available.

```
template<class BASE>
class csr_matrix
    : public term_nesting< interval_iterator<int>,
                           csr_row<BASE> > {
    // ...
};

template<class BASE>
class csr_row
    : public term_nesting< csr_row_iterator<BASE>,
                           term_scalar<BASE> > {
    /// ...
};

template<class BASE>
class csr_row_iterator :
    public offset_iterator<int> {
    // ...
};
```

Figure 5: CSR using the BGML

```
template<class BASE>
class coo_matrix
    : public term_nesting< coo_iterator<BASE>,
                           term_scalar<BASE> > {
    // ...
};

template<class BASE>
class coo_iterator :
    public unordered_iterator< pair<int,int> > {
    // ...
};
```

Figure 6: COO using the BGML

```
// Dense matrix storage
template<class BASE>
class dense_matrix
    : public term_nesting2< interval_iterator<int>,
                            interval_iterator<int>,
                            term_scalar<BASE> > {
    // ...
};
```

Figure 7: Dense using the BGML

4.2 Interfaces for Iterators

The abstract classes for the iterators are described in Figure 4(b).

Iterators in the BGML are used for enumerating indices only. That is, they do not provide methods for accessing the substructures. Instead, the substructures are obtained via the `subterm` method in each `term_nesting` class. This is done, because whenever two independent iterators appear in a level of the index nesting, (e.g., in the dense matrix storage format), the matrix elements are associated with two indices from two different iterators. Since in this case, the value is not associated with a single iterator, it cannot be accessed via a method in either iterator. Thus, the method for accessing the value is placed in the `term_nesting` classes.

In addition to `unordered_iterator`, `increasing_iterator`, and `decreasing_iterator` iterators, we provide the `offset_iterator` interface for iterators whose positions can be randomly accessed, similar to the `random_access_iterator`'s found in the STL. The `interval_iterator` is a refinement of `offset_iterator`, which is used to represent all of the integer indices between a fixed lower and upper bound.

4.3 An Example: Compressed Row Storage

In order to illustrate the use of the BGML, here we present an extended example of its use.

The index structure of the Compressed Sparse Row storage (CSR) in Figure 2 can be described as $r \rightarrow c \rightarrow v$.

Class `csr_matrix` is the top-level class implementing the CSR format. It provides access to the rows of the sparse matrix and corresponds to the $r \rightarrow \cdots$ term in the abstract view. As the array `rowptr` provides random access to a particular row in the matrix, this nesting level is described to the compiler as `interval_iterator`.

```
////////////////////////////////////////////////////
//                  csr_matrix                    //
////////////////////////////////////////////////////

template<class BASE>
class csr_matrix :
    public term_nesting< interval_iterator,
                         csr_row<BASE> >
{
public:
    int * rowptr;
    int * colind;
    BASE * values;
    template<class A_CLASS>
    // Constructors & destructors
    csr_matrix(const proto_term<
                typename A_CLASS::traits_type,
                A_CLASS> &A)
        : own_storage_p(true) { ... }
    ~csr_matrix() { }
    // ...
    // Implementation of the term_nesting< ... >
    // interface
    iterator_type v_tn_begin() const
        { return interval_iterator(0); }
    iterator_type v_tn_end() const
        { return interval_iterator(rows()); }
    subterm_type v_tn_subterm(iterator_type it) {
        int i = *it;
        int jj_lb = rowptr[i], jj_ub = rowptr[i+1];
        return csr_row<BASE>(
            columns(), colind, values, jj_lb, jj_ub);
    }
};
```

The classes `csr_row` and `csr_row_iterator` provide access to the non-zero elements within a row of the CSR matrix. They implement the $c \rightarrow v$ part of the abstract view. The `offset_iterator` tells the compiler that elements within a row are sorted.

```
////////////////////////////////////////////////////
//                    csr_row                     //
////////////////////////////////////////////////////

template<class BASE>
class csr_row
    : public term_nesting<
                csr_row_iterator<BASE>,
                term_scalar<BASE> >
{
public:
```

```
    int * indices;
    BASE * storage;
    int lb; int ub; // ub is not inclusive.
    // Constructors & destructors
    template<class V_CLASS>
    csr_row(const proto_term<
                      typename V_CLASS::traits_type,
                      V_CLASS> &v)
        : own_storage_p(true) { ... }
    ~csr_row() { ... }
    // ...
    // Implementation of the term_nesting< ... > interface
    iterator_type v_tn_begin() const
        { return csr_row_iterator<BASE>(
            indices, storage, lb); }
    iterator_type v_tn_end() const
        { return csr_row_iterator<BASE>(
            indices, storage, ub); }
    subterm_type v_tn_subterm(iterator_type it) {
        return storage[it.jj]; }
};

//////////////////////////////////////////////////
//                csr_row_iterator              //
//////////////////////////////////////////////////

template<class BASE>
class csr_row_iterator :
    public offset_iterator<int>
{
protected:
    int * indices; BASE * storage;
    int jj;
public:
    csr_row_iterator(int *indices = 0,
                     BASE *storage = 0, int jj = -1)
        : indices(indices), storage(storage), jj(jj)
        { }
    // Implementation of the offset_iterator interface.
    const key_type v_deref() const
        { return indices[jj]; }
    bool v_equal(const csr_row_iterator<BASE> &y)
        const
        { return jj == y.jj; }
    void v_set(const csr_row_iterator<BASE> &y)
        { jj = y.jj; }
    int v_distance(const csr_row_iterator<BASE> &y)
        { return jj - y.jj; }
    void v_incr_delta(int d) { jj += d; }
    void v_decr_delta(int d) { jj -= d; }
};
```

After instantiating the generic matrix-vector multiplication program shown in Figure 1 for the CSR format, our compiler produces the C++ code shown below.

```
template <>
void mvm(csr_matrix<double> &A, double x[], double y[])
{
    for (int i = 0; i < A.rows(); i++)
        y[i] = 0.0;

    for (interval_iterator it_r = A.begin();
         it_r != A.end(); it_r++) {
        int r = *it_r;
        csr_row<double> Ar = A.subterm(it_r);
        for (crs_row_iterator<double>
             it_c = Ar.begin();
             it_c != Ar.end(); it_c++) {
            int c = *it_c;
            double v = Ar.subterm(it_c);
            y[r] += v * x[c];
        }
    }
}
```

5 Template Instantiation and Performance

5.1 Implementing for Performance

When we started implementing our generic programming system, we did so in the safest and most straightforward way. This implementation performed abysmally. Several aspects of this implementation that turned out to be pivotal to its performance are discussed below.

Virtual Methods The first is the use of abstract base classes. It is standard practice [25] when programming in C++ to define base classes which contain unimplemented virtual methods. These so-called abstract base classes serve as interfaces in the sense that any class that inherits from them is required to implement all of the virtual methods. Unfortunately, virtual method invocations are generally very expensive. Not only is the overhead of invoking a virtual method high, but the mere presence of a call to a virtual method usually prevents method inlining and subsequent optimization.

```
// Abstract base class
class BASE {
    virtual int mthd1() = 0;
    virtual int mthd2() = 0;
};

// Class to implement BASE interface
class DERIVED : public BASE {
    virtual int mthd1() { ... }
    virtual int mthd2() { ... }
};

void foo (BASE &x, DERIVED &y) {
    // neither of these calls can be resolved without
    // knowing the actual type of x and y.
    ... x.mthd1() ...
    ... y.mthd2() ...
}
```

This problem is discussed in some detail by Veldhuizen [29]. In our new implementation, we chose to use what Veldhuizen calls the "Barton and Nackman trick". In this case, the derived class appears as a template argument to the abstract base class. Then, instead of using virtual method dispatch, the "virtual" methods in the base class cast the `this` pointer to the derived class, and then invoke the appropriate methods directly from the derived class.

```
// Abstract base class
template<class C>
class BASE {
    int mthd1()
      { return static_cast<C&>(*this).mthd1_body(); }
    int mthd2()
      { return static_cast<C&>(*this).mthd2_body(); }
};

// Class to implement BASE interface
class DERIVED : public BASE<DERIVED> {
    int mthd1_body() { ... }
    int mthd2_body() { ... }
};

template<class C>
void foo (BASE<C> &x, DERIVED &y) {
```

```
    // both of these methods can be statically
    // resolved and then inlined.
    ... x.mthd1() ...
    ... y.mthd2() ...
}
```

The `restrict` keyword The potential for aliasing between pointers severely restricts many optimizations that can be performed in all but the most trivial C++ programs. In ANSI/ISO C [10], the keyword `restrict` was introduced to address this problem. When `restrict` is used to qualify a pointer or reference, it roughly means that "no other pointer or reference points to the same memory as this pointer or reference". In other words, nothing can alias a `restrict`'ed pointer.

Even though it is not part of the C++ standard, we use the `restrict` throughout our new implementation. If a compiler does not support this keyword, then we ensure that it is defined to a macro that expands to an empty string.

Runtime decision trees The third aspect important to performance, was the use of "decision tree" style code to control the behavior of the algorithm. The matrix-times-matrix operation in the NIST Sparse BLAS library computes $C \leftarrow \alpha * A * B + \beta * C$. There are several details that the user must specify in order to control the behavior of the basic algorithm,

- whether or not A or A^T is to be used.
- whether the matrices are indexed with 0 (for C/C++) or 1 (for Fortran).

In addition, there are several properties that can be exploited by the algorithm to improve performance,

- If $\alpha = 1$, then scaling by α is a nop, which can be optimized away.
- If B and C are matrices, then a loop is required to visit the elements within their rows. If B and C are vectors, then a single array access may be used for each "row".
- There are additional properties having to do with symmetry and structure that can also be exploited.

When we first wrote the matrix-times-matrix routine, we used a sequence of conditionals "decision tree" in order to identify the particular scenario at hand.

```
void mm(..., double alpha, int offset, ...) {
    if (alpha == 0.)
        if (offset == 0)
            // alpha == 0. && offset == 0 && ...
        else
            // alpha == 0. && offset == 1 && ...
    else if (alpha == 1.)
        if (offset == 0)
            // alpha == 1. && offset == 0 && ...
        else
            // alpha == 1. && offset == 1 && ...
    else
        // only handle 0. and 1. for this example
        assert(false);
}
```

As the number of decisions that we wanted to make increased, the code increased exponentially. The code quickly became difficult to write and maintain. This led us to keep the number of decisions small, which resulted in more general code that was specialized for fewer cases.

In the new implementation, we used templated functions to make the amount of code that we had to write linear in the number of decisions. We did this by first introducing some auxiliary classes that, in effect, encoded constants, such as 0 and 1, as types.

```
template<class BASE>
class spblas_num {
private:
    BASE x;
public:
    spblas_num(BASE x) : x(x) { }
    operator BASE() const { return x; }
};

template<class BASE>
class spblas_zero : public spblas_num<BASE> {
public:
    spblas_zero() : spblas_num<BASE>(0) { }
    operator BASE() const { return 0; }
};

template<class BASE>
class spblas_one : public spblas_num<BASE> {
public:
    spblas_one() : spblas_num<BASE>(1) { }
    operator BASE() const { return 1; }
};
```

Then, we broke our code into a sequence of small procedures, where exactly one decision was made in each procedure. In each procedure, conditionals such as "`alpha == 0.`" result in a value of type "`spblas_zero<double>`" being passed as `alpha` to the next procedure.

```
void mm(..., double alpha, int offset, ...) {
    if (alpha == 0.)
        mm1(..., spblas_zero<double>(), offset, ...)
    else if (alpha == 1.)
        mm1(..., spblas_one<double>(), offset, ...)
    else
        // only handle 0. and 1. for this example
        assert(false);
}

template<class ALPHA>
void mm1(..., ALPHA alpha, int offset, ...)
{
    if (offset == 0)
        mm2(..., alpha, spblas_zero<int>(), ...)
        // alpha == ?? && offset == 0 && ...
    else
        mm2(..., alpha, spblas_one<int>(), ...)
        // alpha == ?? && offset == 1 && ...
}

template<class ALPHA, class OFFSET>
void mm2(..., ALPHA alpha, OFFSET offset, ...)
{
    // just use 'alpha' and 'offset'
}
```

When this code is compiled, four instances of `mm2` are instantiated. In each of these cases, the constant values of

`alpha` and `offset` are available once method inlining is performed. The end result is that we can handle as many cases as we want while only having to write a linear amount of code.

5.2 Existing C++ compilers

We were not terribly happy with the C++ compilers that we used to compile our codes. In many cases, the compiler's conformance to the C++ standard was questionable. The Microsoft Visual C++ compiler (6.0 sp3) [6], for instance, does not correctly scope identifiers declared in the initializer of `for` statements (there is a trick to get around this), nor does it allow static members to be initialized inside of template classes (there is no such trick in this case). Even G++ 2.95.2, the latest version of the GNU C++ compiler [26], comes with a version of the Standard C++ libraries that does not interact well with the use of `namespace` constructs in the user's code.

Even when we could get our code to compile, we found that its performance was usually poor. Many of the compilers that we tested appear not to perform all of the inlining and optimization necessary to obtain the level of performance that a programmer would obtain by writing the code directly in C. It was not uncommon to observe a slowdown of a factor of 3 to 5 in the compiled C++ code. In some cases we even observed a slowdown of 70!

The notable exception to this was Kuck and Associates KCC compiler [14]. Not only did this compiler adhere closely to the standard, but it produced the most efficient code of all of the compilers that we tested.

5.3 Performance

The following table shows the performance of the code from our system compiled using various C++ compilers. The row labeled "g++" refers to GNU g++ 2.95.2, and the row labeled "KCC" refers to KAI KCC 3.4f. The column labeled "BGML" shows the performance of our C++ in megaflops and the column labeled "NIST" shows the performance of equivalent code taken from the NIST C SPBLAS library [19]. Both codes were compiled using the same compiler at the same level of optimization. All codes were run on a 300MHz Pentium II running Redhat Linux 6.1.

	BGML	NIST	ratio
g++	9.87	32.19	0.31
KCC	29.64	31.25	0.95

Figure 8: Performance of C++ vs. handwritten C

We are encouraged to see that the KCC compiler was able to get within 5% of the handwritten code. This indicates to us that our approach and implementation are reasonable and efficient, and that the other compilers have alot of work to do in order to catch up with KCC.

6 Related Work

Generic programming Our work is in the spirit of generic programming which is "the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software" [17]. An important difference from existing generic programming systems is that in our system, the API used in writing generic algorithms is different from the API that is supported by the implementors of compressed formats. *Supporting dual API's effectively requires advanced restructuring compiler technology and can be viewed as a sophisticated form of template instantiation.*

Other researchers have recognized that the level of abstraction of programs can be raised by combining generic programming with more sophisticated compiler technology than is usually available for template instantiation. Our work is close in spirit to that of Batory and co-workers [22, 23] who have used similar ideas in designing the DiSTiL system, a software generator for container data structures. DiSTiL is a declarative language that extends C with constructs for specifying complex data structures declaratively. Data structures are specified by type equations that permit composition of DiSTiL components. When a DiSTiL program is compiled, these declarative specifications are replaced with efficient C implementations by the DiSTiL compiler. DiSTiL's goal is to support standard data structures, not sparse matrices, and no restructuring of code is done during the compilation process.

Aspect-oriented programming The Programmer API presents a simple view of compressed formats that permits programmers to write generic code, but it does not by itself permit the compiler to generate efficient code. the Compiler API conveys additional information about compressed formats to the compiler in order to permit it to generate more efficient code. These additional properties *cross-cut* the `get/set` abstractions of the basic API, and are *aspects* in the terminology of Kiczales [12].

Kiczales and others have designed *aspect-oriented extensions* to Java [15] to permit the expression of such aspects in Java classes in a modular fashion, using compiler technology to exploit aspects for generating efficient code. The key advantage is that resulting programs are simpler to read and maintain because algorithms and aspects are coded separately, and the algorithm is not cluttered with what are essentially implementation details. There are ongoing efforts to write sparse matrix factorization codes using these ideas [11, 18]; however, they do not provide an API for supporting user-defined data structures.

Restructuring compilers Traditionally, restructuring compiler technology has been used to restructure dense matrix programs to enhance parallelism or locality of reference, but it cannot be used directly to restructure sparse matrix programs. This is because program analysis techniques are based on integer linear programming, and can be used only if all array subscripts are affine functions of loop index variables. Such subscripts are common in dense matrix programs in which arrays are accessed by row, column or diagonals, but are the exception in sparse matrix programs since sparse arrays are accessed through indirection arrays.

Bik and Wijshoff at Leiden University were the first to apply restructuring compiler technology to *synthesize* sparse matrix programs from dense matrix programs [5]. Initially their compiler had knowledge of a small number of formats built into it. The formats they considered can be called *Compressed Hyperplane Storage* (CHS) formats since they are obtained by doing a basis transformation on the dense array index space and then compressing out the non-zeros along one or more dimensions. CSR and CSC are therefore special cases of CHS formats. Their compiler analyzed and restructured the input code to match a CHS format, and generated sparse code for that format. More recently, they have developed nonzero structure analysis that supports wider variety of sparse matrix formats [4].

Sparse matrix libraries A number of projects in the numerical analysis community have exploited generic programming to support sparse matrix computations. PETSc [3] is a successful library from Argonne which has a large collection of iterative solvers. These solvers must be linked with user-supplied BLAS that must be written for the particular sparse format of interest. The BLAS are invoked directly by PETSc code, so no special compiler support is needed for PETSc. In contrast, our system permits even the BLAS to be written in a generic, data-structure-neutral fashion, although at the cost of requiring aggressive restructuring compiler technology for generating efficient code.

POOMA [7] and Blitz++ [28] are two more recent packages for matrix computations. The API for both packages is essentially the Programmer API described in this paper. A rich set of C++ templates are provided in both packages, with which a programmer can assemble matrix implementations and produce matrix programs. Some optimizations can be performed by the compiler by relying on Template Expressions [27], but the range of such optimizations is limited, and they can be cumbersome to use. In particular, programmers must provide their own implementations of operations like MVM or triangular solve.

The MTL [21] is another C++ matrix library in which matrices are viewed as containers of containers. This idea is analogous to indexed sequential access, but not as rich as the structures that we discuss in this paper. Also, MTL does not have high- and low-level API's, as we do.

References

[1] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghil. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of SC2000*, Dallas, Texas, November 2000. To appear.

[2] Matthew Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, MA, 1998.

[3] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 – Revision 2.0.15, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[4] Aart Bik and Harry Wijshoff. Automatic nonzero structure analysis. *SIAM Journal of Computing*, 28(5):1576–1587, 1999.

[5] Aart Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 1993 International Conference on Supercomputing*, pages 416–424, Tokyo, Japan, July 20–22, 1993.

[6] Microsoft Corporation. Microsoft Visual C++ home page. `http://msdn.microsoft.com/visualc/`, accessed March 20, 2000.

[7] James A. Crotinger, Julian Cummings, Scott Haney, William Humphrey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy J. Williams. Generic programming in POOMA and PETE. In *Proceedings from Dagstuhl Seminar on Generic Programming*, April 27–May 1, 1998. `http://www.acl.lanl.gov/pooma/papers/GenericProgrammingPaper/dagstuhl.pdf`.

[8] BLAS Techinal Forum. Sparse BLAS library: Lite and toolkit level specifications, January 1997. `http://math.nist.gov/spblas/blast.ps`, Edited by Roldan Pozo and Micheal A. Heroux and Karin A. Remington.

[9] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[10] International Organization for Standards. *ISO/IEC 9899:1999, Programming languages – C*.

[11] John Irwin, Jean-Marc Loingtier, John Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. Technical Report SPL97-007 P9710045, Xerox Palo Alto Research Center, February

1997. http://www.parc.xerox.com/spl/projects/aop/tr-aml.htm.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox Palo Alto Research Center, February 1997. http://www.parc.xerox.com/spl/projects/aop/tr-aop.htm.

[13] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Proceedings of EUROPAR*, 1997. http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/europar97.ps.

[14] Kuck & Associates, Inc. Introduction of KAI C++. http://www.kai.com/C_plus_plus/index.html, accessed March 20, 2000.

[15] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ. In *ECOOP'98 Workshop Reader*, 1998. Springer-Verlag LNCS 1543.

[16] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000.

[17] David R. Musser and Alexander A. Stepanov. Generic programming. In *First International Joint Conference of ISSAC-88 and AAECC-6*, Rome, Italy, July 4-8, 1988. http://www.cs.rpi.edu/~musser/genprog.ps. Also appears in LNCS 358.

[18] William Pugh and Tatiana Shpeisman. Generation of efficient code for sparse matrix computations. In *The Eleventh International Workshop on Languages and Compilers for Parallel Computing*, LNCS, Springer-Verlag, Chapel Hill, NC, August 1998.

[19] Karen A. Remington. The NIST Sparse BLAS (v. 0.9) sparse matrix computational kernels, January 9, 1997. http://math.nist.gov/spblas/, Accessed March 20, 2000.

[20] Yousef Saad. SPARSKIT version 2.0. http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html.

[21] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE '98*, 1998. http://www.lsc.nd.edu/downloads/research/mtl/papers/iscope_final.ps.gz.

[22] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *1997 Usenix Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. ftp://ftp.cs.utexas.edu/pub/predator/distil.ps.

[23] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *ECOOP '98*, July 1998. ftp://ftp.cs.utexas.edu/pub/predator/templates.ps.

[24] Paul Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, July 1997. http://cs-tr.cs.cornell.edu/Dienst/UI/1.0/Display/ncstrl.cornell/TR97-1635. Also as Technical Report CORNELLCS:TR97-1635.

[25] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition, 1997.

[26] The GCC Team. GCC home page. http://gcc.gnu.org/, accessed March 20, 2000.

[27] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. http://monet.uwaterloo.ca/~tveldhui/papers/Expression-Templates/exprtmpl.html.

[28] Todd Veldhuizen. The Blitz++ array model. In *ISCOPE '98*, 1998. http://monet.uwaterloo.ca/blitz/papers/iscope98.ps.

[29] Todd Veldhuizen. Techniques for scientific C++, version 0.3, August 1999. http://extreme.indiana.edu/~tveldhui/papers/techniques/, accessed March 20, 2000.